Graph Store Proposal
~2020.06.15
Logan Allen

[Demo]

Rationale
Graph Store is a continuation of the store / hook model, but iterates on the model by moving away from a proliferation of application-specific stores. Instead, Graph Store seeks to be an example of a consistent, strongly validated, interoperable data storage format that can reduce the amount of bespoke application code by about 6000 lines in `/app`, enable faster iteration time by the Interface team, and reify Airlock by allowing client developers to write a variety of performant, flexible applications for Urbit without a need to write Hoon.

Architecture

A review of current app architectures, their strengths and weaknesses, and how Graph Store iterates on the existing models:

How does Publish work?

As an experiment, Publish explores how to build Gall applications that orient their entire data flow around writing data into Clay and reading data out of Clay.

Publish has a container data structure called a notebook, which is represented by a folder within Clay and a metadata file. A notebook stores metadata about the posts within it, and has subfolders that each contain a body text file and a folder of comments.

Publish is novel in that no other applications have attempted to heavily utilize Clay to the same degree. As a result, Publish makes Unix text editing and data export into first class flows, a unique property amongst current Urbit applications, as well as storing data in a consistent and interoperable format. Weaknesses of Publish are that its dependence for all its life cycles on Clay introduces asynchronicity, performance consequences, extra parsing steps, and less flexible data queries.

Publish as a singular monolithic application does not utilize the store / hook model, as it uses Clay as its data store, and includes all of the logic and wrappers for Clay interactions as well as all networking logic in a singular agent.

The lifecycle of a new Publish post is as follows:

- Receive post poke
- Write file to Clay (asynchronous)
- Read file out of clay (asynchronous)
- Parse file
- Process effects of new post (send diffs)
- Parse effects into JSON for web UI

How does Chat work?

Chat was the first application experiment with the store / hook model. The primary rationale of the store / hook experiment was to reduce complexity within applications by allowing a store agent to function as a reactively queryable database table, and a hook agent to handle network events and replication. As this architecture model was experimental at the time, the data structures within chat were made as simple as possible to act as a pure test of the model.

Chat uses Gall agent state as its data storage model, and has a container data structure called a mailbox, which is a tuple within a map of unique identifiers to mailboxes. The tuple contains a list of envelopes, which are messages.

Strengths of this model are that it is simple to reason about and has close to optimal performance characteristics. Weaknesses have been that data export would require bespoke tooling built within the application and data is stored in an application-specific data structure which encourages the proliferation of more bespoke state machines for applications as opposed to storing data in legible and consistently interoperable formats.

A weakness of the chat design is that in optimizing for simplicity, it does no data validation in its replication scheme. This has led to issues in which messages can be dropped or duplicated in edge cases such as during complex OTAs. The chat data structure is append-only, and does not allow insertion of envelopes at arbitrary indices. This has led to an increased complexity within the Chat CLI which seeks to preserve the feature of being able to view particular messages by index in a long-form terminal reader and thus expects a stronger ordering guarantee than the Chat Store provides.

The lifecycle of a chat message is as follows:

- Receive post poke
- Save in Gall state
- Process effects of new post (send diffs)
- Parse effects into JSON for web UI

How does the Graph Store work?

Graph Store is a continuation of the store / hook model, but iterates on the model by moving away from a proliferation of application-specific stores. Instead, Graph Store seeks to be an example of a consistent, strongly validated, interoperable data storage format that is primarily informed by the traits of the data being stored.

Graph Store uses Gall agent state as its data storage model, and has a container data structure called a graph, which is an ordered map within a map of unique identifiers to graphs.

The graph, an ordered map, contains nodes. Simplified a bit for brevity, a node is:

```
      +$  node  [=post children=(unit graph)]
```

This allows for infinitely recursive children, which could occupy different semantics within different applications (annotations on a book at a particular page, comments on a blog, etc). Thus, a graph is in fact a tree, in which each child has a singular parent. As an aside, children may reference any node at any index the graph, including parents, but the graph data structure is best thought of as a tree rather than a fully unconstrained graph with fully realized cycles.

Graph Store primarily follows as an iteration of Chat's model that attempts to drastically improve the expressiveness of the data structure to rival the best parts of Publish. Graph Store retains the promising performance characteristics of Chat by utilizing Gall agent state and the ordered map structure, which provides ~O(2 * log n) subset retrieval, O(log n) item retrieval, and O(log n) insertion. With Liam's data export work, one of the main weaknesses of Chat is resolved, in that data export can be consistently performed without bespoke application code to support it.

With the Graph Store, additional agent data stores will only be necessary for configuration details, "table joins", or for queryable structures that have vastly different characteristics than most social media we interact with today. As an aside, one can imagine an eventual desire for a SQL-like relational store, or a DAG store in the future.

Graph Store implements strong, flexible validation upon graph shape using the mark system, allowing one to specify a mark that constrains the data allowed into their graph.

Graph Store solves Chat's lack of strong-ordering semantics via its ordered map structure, and includes strong validation on data within graphs by storing a hash of the parent node of a child, the contents of a post, the author, the time sent, and signing that data with the private key of the authoring ship.

The lifecycle of a post to a graph is as follows:

- Receive post poke
- Validate contents of post
- Save in Gall state
- Process effects of new post (send diffs)

- Parse effects into JSON for web UI

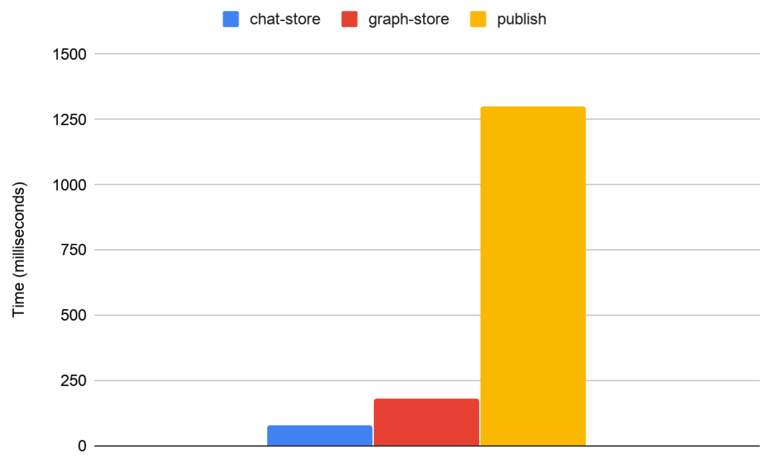
Performance Tests

Expectations: chat is marginally faster than graph and both chat and graph are significantly faster than publish

Remember to put manual graph poke parsing into gall for the duration of tests for a true comparison (or remove the chat and publish hack)

Tests:

- How long does an event to process a new post take? (without the Ford hack for faster build time but *with* a +poke-json to optimize performance)



- How long does it take to load a truncated chat inbox when chat has 20 chatrooms and 10,000 messages per room?
- How long does it take to load a truncated graph page when there are 20 graphs and 10,000 messages per graph?
- How long does it take Publish to load when there are 20 blogs and 10,000 posts per blog?

Integration Plan

Phase One:

1. Write Post as new standalone microblogging product that has much of the same functionality as Publish.
2. Test and polish.
3. Release OTA
4. Solve any bugs that come up.

Phase Two:

1. Write migration scripts for Chat, Links, and Publish to migrate data into graph and set a flag to stop accepting new pokes and for their hooks to stop accepting updates.
2. Transition Chat, Links, and Publish interfaces to check if data has been migrated to graph yet and if not say “please run script to migrate data
3. Transition chat-cli to work with graphs and stop storing posts.
4. Set up tracking instead of sync for OTAs and auto-migrate any remaining data.
5. Release OTA.

Eventually:

- Write a “graph cli” for users to write posts, replies, view posts, etc. Do not store any posts within it. Allow seeing parents of posts, posts and replies, and to turn “update printing” in the CLI on or off